Зміст

[1. Завдання (варіант 17) 2](#_Toc185519962)

[2. Опис програмного рішення 4](#_Toc185519963)

[3. Дослідження надійності 11](#_Toc185519964)

[3.1 Початкова схема 14](#_Toc185519965)

[3.2 Модифікація 1: 7-7-7-8-8 16](#_Toc185519966)

[3.3 Модифікація 2: з’єднання елементів 18](#_Toc185519967)

[3.4 Модифікація 3: 24 елементи (d9) 19](#_Toc185519968)

[3.5 Модифікація 4: 25 елементів (d10) 21](#_Toc185519969)

[3.6 Модифікація 5: 27 елементів (c7, c8) 23](#_Toc185519970)

[3.7 Модифікація 6: 29 елементів (a3, a4) 25](#_Toc185519971)

[4. Додаткове тестування швидкодії 28](#_Toc185519972)

# 1. Завдання (варіант 17)

Для виконання лабораторної роботи було обрано варіант 17:

A diagram of a triangle

Description automatically generated

рис.1 Схема варіант 17

Складові логічної функції F f1,f2,f3,f4,f5,f6:

Таблиця реконфігурації (Ln – номінальне навантаження, Lm – максимальне навантаження):

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ln | Lm | Перерозподіли навантаження |
| pr1 | 50 | 80 | [pr2 = 25, pr3 = 25] |
| pr2 | 50 | 80 | [pr1 = 25, pr3 = 25] |
| pr3 | 50 | 80 | [pr1 = 25, pr2 = 25] |
| pr5 | 30 | 60 | відсутні |
| pr6 | 30 | 60 | відсутні |

Таблиця вказує перерозподіли:

Якщо pr1 відмовив фізично – 25 одиниць навантаження додаються до pr2, 25 одиниць навантаження додаються до pr3.

Якщо pr2 відмовив фізично – 25 одиниць навантаження додаються до pr1, 25 одиниць навантаження додаються до pr3.

Якщо pr3 відмовив фізично – 25 одиниць навантаження додаються до pr1, 25 одиниць навантаження додаються до pr2.

Для процесорів pr5, pr6 перерозподіли відсутні.

# 2. Опис програмного рішення

Для автоматизації розрахунків було створено програму мовою С++, що обчислює надійність системи. Посилання на github репозиторій: [Bohdan628318ylypchenko/SchemeReliability: Scheme reliability lab by Bohdan Pylypchenko and Ihor Bezrukov](https://github.com/Bohdan628318ylypchenko/SchemeReliability/tree/master).

Ключові особливості алгоритму та реалізації:

* Програма оброблює ВСІ можливі початкові вектори станів системи (для варіанту 17 кількість всіх можливих станів = ). Для здійснення обчислень за прийнятний час програма використовує паралелізм: головний потік розподіляє вектори станів системи між потоками-обробниками згідно принципу “round robin” (подібно до того як карти розподіляються між гравцями у покері).
* Програма зберігає результати обчислень як набір .ssv файлів. У файлах зберігаються:
  + Початковий стан системи (F(sv1)).
  + Стан системи після реконфігурації (F(sv2)).
  + Імовірність виникнення стану sv1.
  + Значення «координат» вектору sv1.
  + Значення «координат» вектору sv2.
* Для більш глибокого аналізу надійності системи було створено скрипт мовою python (+ pandas). На основі .ssv файлів скрипт обчислює метрики:
  + Імовірність відмови для кожного елементу: сума імовірностей всіх sv1, у яких даний елемент відмовив.
  + Кількість відмов для кожного елементу.
  + Імовірність роботоздатності / відмови системи (P, Q) в якості додаткової перевірки правильності розрахунків основної програми.
* Також скрипт перевіряє, чи виникли ситуації, коли F(sv1) = 1 але F(sv2) = 0. Якщо така ситуація відсутня, реконфігурація є консистентною. В противному випадку конфігурація є неконсистентною.

Схема загального алгоритму обчислення надійності, що реалізований програмою, наведена на рис. 2.

A diagram of a flowchart

Description automatically generated

Рис. 2 Алгоритм обчислення надійності системи

У роботі реалізовано 2 алгоритми реконфігурації:

* Brute force (майже повний перебір всіх комбінацій перерозподілів).
* Жадібний алгоритм.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ln | Lm | Перерозподіли навантаження |
| pr1 | 50 | 100 | [pr2 = 40, pr3 = 10], [pr2 = 30, pr3 = 20], [pr2 = 25, pr3 = 25], [pr2 = 20, pr3 = 30], [pr2 = 10, pr3 = 40], [pr2 = 50], [pr3 = 50] |
| pr2 | 50 | 100 | [pr1 = 40, pr3 = 10], [pr1 = 30, pr3 = 20], [pr1 = 25, pr3 = 25], [pr1 = 20, pr3 = 30], [pr1 = 10, pr3 = 40], [pr1 = 50], [pr3 = 50] |
| pr3 | 50 | 100 | [pr1 = 40, pr2 = 10], [pr1 = 30, pr2 = 20], [pr1 = 25, pr2 = 25], [pr1 = 20, pr2 = 30], [pr1 = 10, pr2 = 40], [pr1 = 50], [pr2 = 50] |
| pr5 | 30 | 60 | [pr6 = 30], [pr1 = 20, pr2 = 20, pr3 = 20], [pr1 = 25, pr2 = 25], [pr2 = 25, pr3 = 25], [pr1 = 25, pr3 = 25], [pr1 = 30], [pr2 = 30], [pr3 = 30] |
| pr6 | 30 | 60 | [pr5 = 30], [pr1 = 20, pr2 = 20, pr3 = 20], [pr1 = 25, pr2 = 25], [pr2 = 25, pr3 = 25], [pr1 = 25, pr3 = 25], [pr1 = 30], [pr2 = 30], [pr3 = 30] |

Таблиця реконфігурацій задає набір можливих переходів (перерозподілів) навантажень для процесорів системи. Згідно таблиці вище, для процесорів pr1-3 маємо по 7 варіантів перерозподілів, для процесорів 5, 6 – по 8

Вважатимемо перерозподіл **можливим**, якщо всі процесори, на які відводиться навантаження, є робочими.

Вважатимемо перерозподіл **успішним**, якщо одночасно виконуються дві умови:

1. перерозподіл є можливим
2. навантаження на процесори перерозподілу менші за відповідні максимальні значення.

Нехай відмовили процесори 1, 3, 6. Тоді, згідно таблиці реконфігурації, існує 7 \* 7 \* 8 = 392 повних варіантів реконфігурації (комбінацій перерозподілів), 7 \* 7 + 2 \* 7 \* 8 часткових реконфігурацій розмірності 2, 7 + 7 + 8 реконфігурацій розмірності 1. Разом 575 варіантів.

Ключові властивості brute force алгоритму:

* В основі алгоритму лежить обхід дерева вглибину за допомогою рекурсії.
* Обхід дерева зупиняється передчасно, якщо знайдено реконфігурацію, для якої F(sv2) = 1.
* Якщо не існує такої реконфігурації, що F(sv2) = 1, алгоритм обиратиме реконфігурацію із максимальною кількістю робочих елементів.
* Якщо перерозподіл, що розглядається, не є можливим (внаслідок виборів на попередніх рівнях дерева), алгоритм переходить на наступний рівень дерева, «перестрибуючи» перерозподіли поточного рівня. Такі «стрибки» є причиною появи «часткових» реконфігурацій. При цьому після того, як алгоритм дійде до останнього рівня дерева, обхід повернеться до пропущеного рівня дерева і перейде до інших перерозподілів пропущеного рівня.
* В найгіршому випадку алгоритм перебиратиме всі варіанти реконфігурації.

Схема brute force алгоритму зображена на рис. 3:

A diagram of a flowchart

Description automatically generated

Рис. 3 Схема brute force алгоритму

Очевидно, алгоритм повного обходу дерева є повільним. З’являється потреба у алгоритмі, який, можливо, не знаходитиме найкращий варіант реконфігурації, але знаходитиме реконфігурацію задовільної якості за набагато менший час.

Введемо оцінку навантаження J:

де:

кількість елементів системи

– вектор навантаження

вектор максимальних навантажень

функція Гевісайда:

штрафний коефіцієнт. В роботі значення рівне .

Чим меншою є оцінка, тим кращим вважається навантаження.

На основі оцінки J було створено жадібний алгоритм:

A diagram of a flowchart

Description automatically generated

Рис.4 Схема жадібного алгоритму

Для кожного процесору, що потребує реконфігурації, алгоритм оцінює всі **можливі** (можливі у сенсі вищенаведеного визначення) перерозподіли, обирає найкращий. Таким чином для відмов pr1, pr3, pr6 замість 575 варіантів реконфігурацій жадібному алгоритму потрібно оцінити 7 + 7 + 8 = 22 перерозподіли. Іншими словами, жадібний алгоритм дозволив «позбутись» вибухової складності, спричиненої основним правилом комбінаторики.

# Дослідження надійності

Через великі об’єми обчислень лабораторна робота виконувалась на AWS EC2, тип віртуальної машини – c7i.8xlarge (16 ядер / 32 потоки, 64 GB ddr5):

A computer screen shot of a computer

Description automatically generated

Рис. 5 Процесор, використаний для обчислень

Повний перелік характеристик vm можна переглянути у файлі ec2-specs.txt у репозиторії.

Вивід програми для всіх модифікацій схем (одиниця виміру часу – секунда):

PS C:\Users\Administrator\SchemeReliability\x64\Release> .\sr-research.exe

=== original s23 ===

Scheme type = greedy

time = 3

path = s23-original-greedy

sp = 0.9990208569798796, sq = 0.000979143020119051

state count = 8388608

Scheme type = brute

time = 3

path = s23-original-brute

sp = 0.9990208569798796, sq = 0.000979143020119051

state count = 8388608

=== s23 with rt (7 7 7 8 8) ===

Scheme type = greedy

time = 2

path = s23-77788-greedy

sp = 0.9992607082975051, sq = 0.0007392917024915616

state count = 8388608

Scheme type = brute

time = 18

path = s23-77788-brute

sp = 0.9992607082975051, sq = 0.0007392917024915616

state count = 8388608

=== s23 with rt and modified connections ===

Scheme type = greedy

time = 3

path = s23-77788-modified-connections-greedy

sp = 0.999737721521635, sq = 0.0002622784783607316

state count = 8388608

Scheme type = brute

time = 13

path = s23-77788-modified-connections-brute

sp = 0.999737721521635, sq = 0.0002622784783607316

state count = 8388608

=== s24 with d9 right ===

Scheme type = greedy

time = 1

path = s24-d9-right-greedy

sp = 0.9997597157515095, sq = 0.00024028424848725232

state count = 16777216

Scheme type = brute

time = 35

path = s24-d9-right-brute

sp = 0.9997597157515095, sq = 0.00024028424848725232

state count = 16777216

=== s25 with d9 d10 right ===

Scheme type = greedy

time = 8

path = s25-d9-d10-right-greedy

sp = 0.9997597162353798, sq = 0.00024028376461419955

state count = 33554432

Scheme type = brute

time = 61

path = s25-d9-d10-right-brute

sp = 0.9997597162353798, sq = 0.00024028376461419955

state count = 33554432

=== s27 with d9 d10 c7 right c8 left ===

Scheme type = greedy

time = 77

path = s27-d9-d10-c7-right-c8-left-greedy

sp = 0.9997598842949585, sq = 0.00024011570500589362

state count = 134217728

Scheme type = brute

time = 220

path = s27-d9-d10-c7-right-c8-left-brute

sp = 0.9997598842949585, sq = 0.00024011570500589362

state count = 134217728

=== s29 with d9 d10 c7 right c8 left a4 ===

Scheme type = greedy

time = 397

path = s29-d9-d10-c7-right-c8-left-a4-greedy

sp = 0.999999841063663, sq = 1.5893623277066473e-07

state count = 536870912

Scheme type = brute

time = 1022

path = s29-d9-d10-c7-right-c8-left-a4-brute

sp = 0.999999841063663, sq = 1.5893623277066473e-07

state count = 536870912

PS C:\Users\Administrator\SchemeReliability\x64\Release>

## 3.1 Початкова схема

A diagram of a triangle

Description automatically generated

Рис. 6 Оригінальна схема

Опис логічної функції:

Таблиця реконфігурації:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ln | Lm | Перерозподіли навантаження |
| pr1 | 50 | 80 | [pr2 = 25, pr3 = 25] |
| pr2 | 50 | 80 | [pr1 = 25, pr3 = 25] |
| pr3 | 50 | 80 | [pr1 = 25, pr2 = 25] |
| pr5 | 30 | 60 | відсутні |
| pr6 | 30 | 60 | відсутні |

Результати виконання програми:

A computer screen with white text

Description automatically generated

Метрики (жадібний алгоритм ліворуч, повний обхід дерева - праворуч):

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

1. Результати алгоритмів є однаковими.
2. Маємо надійність, що є меншою за 0.9999. Поставимо за мету досягнути надійність, більшу або рівну 0.9999.
3. Реконфігурація очікувано зменшила імовірність відмови процесорів 1, 2, 3.
4. Імовірність відмови процесорів 5 і 6 очікувано не змінилась – перерозподіли для цих процесорів відсутні у таблиці реконфігурації.
5. Час виконання обох алгоритмів – 3 секунди. Схоже розмірність обчислень і мала кількість варіантів реконфігурації для початкової схеми є недостатньо великою щоб різниця між алгоритмами була помітна.

## 3.2 Модифікація 1: 7-7-7-8-8

Модифікуємо таблицю реконфігурації (додамо перерозподіли, збільшимо максимальне навантаження для процесорів 1, 2, 3 із 80 до 100):

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ln | Lm | Перерозподіли навантаження |
| pr1 | 50 | 100 | [pr2 = 50], [pr3 = 50], [pr2 = 25, pr3 = 25], [pr2 = 25, pr5 = 30], [pr2 = 25, pr6 = 30], [pr3 = 25, pr5 = 30], [pr3 = 25, pr6 = 30] |
| pr2 | 50 | 100 | [pr1 = 50], [pr3 = 50], [pr1 = 25, pr3 = 25], [pr1 = 25, pr5 = 30], [pr1 = 25, pr6 = 30], [pr3 = 25, pr5 = 30], [pr3 = 25, pr6 = 30] |
| pr3 | 50 | 100 | [pr1 = 50], [pr2 = 50], [pr1 = 25, pr2 = 25], [pr1 = 25, pr5 = 30], [pr1 = 25, pr6 = 30], [pr2 = 25, pr5 = 30], [pr2 = 25, pr6 = 30] |
| pr5 | 30 | 60 | [pr6 = 30], [pr1 = 35], [pr2 = 35], [pr3 = 35], [pr1 = 18, pr2 = 18], [pr2 = 18, pr3 = 18], [pr1 = 18, pr3 = 18], [pr1 = 12, pr2 = 12, pr3 = 12] |
| pr6 | 30 | 60 | [pr5 = 30], [pr1 = 35], [pr2 = 35], [pr3 = 35], [pr1 = 18, pr2 = 18], [pr2 = 18, pr3 = 18], [pr1 = 18, pr3 = 18], [pr1 = 12, pr2 = 12, pr3 = 12] |

Інші параметри схеми залишимо без змін.

Результати виконання програми:

A screenshot of a computer program

Description automatically generated

Метрики (жадібний алгоритм ліворуч, повний обхід дерева - праворуч):

A screenshot of a computer

Description automatically generated A screenshot of a computer program

Description automatically generated

1. Імовірність безвідмовної роботи системи збільшилась: 0.9992 проти 0.9990.
2. Результати реконфігурації відрізняються: алгоритм повного обходу знаходить комбінації перерозподілів, у яких кількість робочих процесорів є більшою у порівнянні з реконфігураціями, знайденими жадібним алгоритмом. При цьому різниця є несуттєвою, адже не вплинула на кінцеве значення імовірності безвідмовної роботи системи.
3. Помітна різниця у складності алгоритмів: жадібний алгоритм закінчив обчислення за 2 секунди, для повного обходу знадобилось 18 секунд.

## 3.3 Модифікація 2: з’єднання елементів

A diagram of a diagram

Description automatically generated

Рис. 7 Схема з модифікованими з’єднаннями

Утворимо додаткові з’єднання (на рисунку позначені червоним кольором).

Внаслідок нових з’єднань логічна функція матиме вид:

Інші параметри залишимо без змін.

Опис нової логічної функції:

.scheme\_function = [](const StateVectorDto<all\_count, processor\_count>& sv)

{

span<bool> s = sv.all;

bool f1 = (s[16] + s[17] + s[18] + s[19]) \* (s[11] + s[12] + s[13]) \* (s[7] + s[8]);

bool f3 = s[0] \* s[1] \* s[2];

bool f4 = s[20] \* (s[14] + s[15]) \* (s[9] + s[10]);

bool f5 = s[3] \* s[4];

bool f6 = s[5] \* s[6] \* (s[21] + s[22]);

return f1 \* f3 \* f4 \* f5 \* f6;

}

Результати виконання програми:

A computer screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedМетрики (жадібний алгоритм ліворуч, повний обхід дерева – праворуч):

1. Імовірність безвідмовної роботи збільшилась: 0.9997 проти 0.9992.
2. Результати реконфігурації відрізняються: алгоритм повного обходу знаходить комбінації перерозподілів, у яких кількість робочих процесорів є більшою у порівнянні з реконфігураціями, знайденими жадібним алгоритмом. При цьому різниця є несуттєвою, адже не вплинула на кінцеве значення імовірності безвідмовної роботи системи.
3. Жадібний алгоритм знову демонструє перевагу у швидкодії: 2 секунди проти 13 секунд.

## 3.4 Модифікація 3: 24 елементи (d9)

Судячи із обчислених імовірностей відмов кожного елементу, реконфігурація успішно компенсує фізичні відмови процесорів, головним джерелом відмов є контролери. Тому для подальшого збільшення надійності необхідно дублювати існуючі методи паралельним з’єднанням.

A diagram of a diagram

Description automatically generated

Рис. 8 Схема з новим елементом d9 (всього 24 елементи)

Додавання нового елементу d9 відобразиться на виді логічної функції:

Інші параметри залишимо без змін.

Результати роботи програми:

A computer screen with white text

Description automatically generated

Метрики:

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. Надійність незначно збільшилась: 0.99975 проти 0.99973
2. Бачимо суттєву перевагу швидкодії жадібного алгоритму: 1 секунда проти 35. При цьому кінцеве значення надійності є однаковим для обох алгоритмів.

## 3.5 Модифікація 4: 25 елементів (d10)

У лівій частині схеми кількість контролерів типу d є більшою за кількість контролерів типу с на 1. Додамо ще один контролер типу d (d10), щоб права частина схеми також мала різницю між кількістю контролерів c і d рівну 1:

A diagram of a diagram

Description automatically generated

Рис. 9 Схема з новим елементом d10 (всього 25 елементів)

Додавання нового елементу d10 відобразиться на виді логічної функції:

Інші параметри залишимо без змін.

Результати роботи програми:

A computer screen with white text

Description automatically generated

Метрики:

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. Надійність майже не підвищилась: 0.99975971623 проти 0.99975971575.
2. Знову спостерігаємо перевагу швидкодії жадібного алгоритму: 8 секунд проти 61. При цьому кінцеве значення надійності є однаковим для обох алгоритмів.

## 3.6 Модифікація 5: 27 елементів (c7, c8)

Додамо контролери c7 та c8 до правої та лівої частини схеми:

A diagram of a diagram

Description automatically generated

Рис. 10 Схема з елементами c7 та c8 (всього 27 елементів).

Зміни функцій та :

Інші параметри залишимо без змін.

Результати роботи програми:

A computer screen shot of a program

Description automatically generated

Метрики:

1. A screenshot of a computer

   Description automatically generatedA screenshot of a computer program

   Description automatically generatedНадійність незначно підвищилась: 0.99975988 проти 0.99975971
2. Жадібний алгоритм очікувано швидший: 77 секунд проти 220

## 3.7 Модифікація 6: 29 елементів (a3, a4)

A diagram of a circuit diagram

Description automatically generated

Рис. 11 Схема з елементами a3, a4 (всього 29 елементів)

Жодна із змін не впливала на контролери типу а. Саме контролери типу а, згідно метрик, є головною причиною відмови всієї схеми. Додамо 2 контролери: a3 та а4. Тепер права та ліва частини схеми мають декілька з’єднань з магістралями m1, m2.

Логічна функція прийме вид:

Інші параметри залишило без змін.

Результати роботи програми:

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generatedМетрики:

1. Спостерігаємо значний приріст надійності: 0.9999998 проти 0.99975. Початкова мета: досягнути надійності, більшої або рівної 0.9999 – досягнута.
2. Імовірність відмов контролерів типу а зменшилась з четвертого до восьмого порядку.
3. Вкотре жадібний алгоритм знаходить варіанти реконфігурацій, які дозволяють досягати кінцевою надійності, ідентичної до повного перебору, за значно менший час.

# Додаткове тестування швидкодії

Обчислення розділу 3 проводились на EC2 vm. Однією з помилок налаштування vm було використання мережевих дисків (EBS volumes). Через це результати порівняння швидкодії було спотворено потенційним впливом продуктивності мереж.

Незначний приріст надійності для модифікацій 4, 5 на кінцеву надійність системи, та суттєвий приріст надійності внаслідок додавання контролерів а3, а4, наводить на думку, що можна отримати надійність, схожу з модифікацією 6, не застосовуючи модифікації 4, 5:

A diagram of a diagram of a system

Description automatically generated with medium confidence

Рис.12 Модифікація 7 (26 елементів)

Логічна функція схеми:

Опис схеми в коді:

constexpr size\_t all\_count { 26 };

constexpr size\_t processor\_count { 5 };

SchemeDto<all\_count, processor\_count> scheme

{

.elements =

{

ElementDto { .name = "a1", .p = ppa, .q = qpa },

ElementDto { .name = "a2", .p = ppa, .q = qpa },

ElementDto { .name = "b1", .p = ppb, .q = qpb },

ElementDto { .name = "b2", .p = ppb, .q = qpb },

ElementDto { .name = "b4", .p = ppb, .q = qpb },

ElementDto { .name = "b5", .p = ppb, .q = qpb },

ElementDto { .name = "c1", .p = ppc, .q = qpc },

ElementDto { .name = "c2", .p = ppc, .q = qpc },

ElementDto { .name = "c4", .p = ppc, .q = qpc },

ElementDto { .name = "c5", .p = ppc, .q = qpc },

ElementDto { .name = "c6", .p = ppc, .q = qpc },

ElementDto { .name = "d1", .p = ppd, .q = qpd },

ElementDto { .name = "d2", .p = ppd, .q = qpd },

ElementDto { .name = "d3", .p = ppd, .q = qpd },

ElementDto { .name = "d6", .p = ppd, .q = qpd },

ElementDto { .name = "d8", .p = ppd, .q = qpd },

ElementDto { .name = "m1", .p = ppm, .q = qpm },

ElementDto { .name = "m2", .p = ppm, .q = qpm },

ElementDto { .name = "d9", .p = ppd, .q = qpd },

ElementDto { .name = "a3", .p = ppa, .q = qpa },

ElementDto { .name = "a4", .p = ppa, .q = qpa }

},

.processors =

{

ProcessorDto

{

.name = "pr1", .p = ppr, .q = qpr, .normal\_load = 50, .max\_load = 100,

.transitions =

{

{ TrUnit { 1, 50 } },

{ TrUnit { 2, 50 } },

{ TrUnit { 1, 25 }, TrUnit { 2, 25 } },

{ TrUnit { 1, 25 }, TrUnit { 3, 30 } },

{ TrUnit { 1, 25 }, TrUnit { 4, 30 } },

{ TrUnit { 2, 25 }, TrUnit { 3, 30 } },

{ TrUnit { 2, 25 }, TrUnit { 4, 30 } }

}

},

ProcessorDto

{

.name = "pr2", .p = ppr, .q = qpr, .normal\_load = 50, .max\_load = 100,

.transitions =

{

{ TrUnit { 0, 50 } },

{ TrUnit { 2, 50 } },

{ TrUnit { 0, 25 }, TrUnit { 2, 25 } },

{ TrUnit { 0, 25 }, TrUnit { 3, 30 } },

{ TrUnit { 0, 25 }, TrUnit { 4, 30 } },

{ TrUnit { 2, 25 }, TrUnit { 3, 30 } },

{ TrUnit { 2, 25 }, TrUnit { 4, 30 } }

}

},

ProcessorDto

{

.name = "pr3", .p = ppr, .q = qpr, .normal\_load = 50, .max\_load = 100,

.transitions =

{

{ TrUnit { 0, 50 } },

{ TrUnit { 1, 50 } },

{ TrUnit { 0, 25 }, TrUnit { 1, 25 } },

{ TrUnit { 0, 25 }, TrUnit { 3, 30 } },

{ TrUnit { 0, 25 }, TrUnit { 4, 30 } },

{ TrUnit { 1, 25 }, TrUnit { 3, 30 } },

{ TrUnit { 1, 25 }, TrUnit { 4, 30 } }

}

},

ProcessorDto

{

.name = "pr4", .p = ppr, .q = qpr, .normal\_load = 30, .max\_load = 60,

.transitions =

{

{ TrUnit { 4, 30 } },

{ TrUnit { 0, 35 } },

{ TrUnit { 1, 35 } },

{ TrUnit { 2, 35 } },

{ TrUnit { 0, 18 }, TrUnit { 1, 18 } },

{ TrUnit { 1, 18 }, TrUnit { 2, 18 } },

{ TrUnit { 0, 18 }, TrUnit { 2, 18 } },

{ TrUnit { 0, 12 }, TrUnit { 1, 12 }, TrUnit { 2, 12 } }

}

},

ProcessorDto

{

.name = "pr5", .p = ppr, .q = qpr, .normal\_load = 30, .max\_load = 60,

.transitions =

{

{ TrUnit { 3, 30 } },

{ TrUnit { 0, 35 } },

{ TrUnit { 1, 35 } },

{ TrUnit { 2, 35 } },

{ TrUnit { 0, 18 }, TrUnit { 1, 18 } },

{ TrUnit { 1, 18 }, TrUnit { 2, 18 } },

{ TrUnit { 0, 18 }, TrUnit { 2, 18 } },

{ TrUnit { 0, 12 }, TrUnit { 1, 12 }, TrUnit { 2, 12 } }

}

}

},

.scheme\_function = [](const StateVectorDto<all\_count, processor\_count>& sv)

{

span<bool> s = sv.all;

bool f1 = (s[16] + s[17] + s[18] + s[19]) \* (s[11] + s[12] + s[13]) \* (s[7] + s[8]);

bool f3 = s[0] \* s[1] \* s[2];

bool f4 = (s[20] + s[23]) \* (s[14] + s[15]) \* (s[9] + s[10]);

bool f5 = s[3] \* s[4];

bool f6 = (s[5] + s[24]) \* (s[6] + s[25]) \* (s[21] + s[22]);

return f1 \* f3 \* f4 \* f5 \* f6;

}

};

scheme.scheme\_name = "s26-final-greedy";

scheme.type = SchemeType::Greedy;

Utils::process\_scheme(scheme);

scheme.scheme\_name = "s26-final-brute";

scheme.type = SchemeType::Brute;

Utils::process\_scheme(scheme);

}

Для додаткового тестування швидкодії виконаємо обчислення для оригінальної схеми, модифікацій 1, 2, 3, 7. Обчислення здійснюватимемо на процесорі:

A computer screen shot of a computer

Description automatically generated

Результати виконання програми:

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generatedДодатково обчислимо метрики для модифікації 7:

1. Різниця у швидкодії жадібного алгоритму та повного обходу стає ще більш очевидною.
2. Надійність модифікації 7 майже не поступається надійності модифікації 6. При цьому схема 7 має на 3 елементи менше ніж схема 6.